Creating production-ready ML pipelines on AWS

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Creating production-ready ML pipelines on AWS

Table of Contents

Creating production-ready ML pipelines on AWS

Chen Wu and Josiah Davis, Amazon Web Services

Machine learning (ML) projects require a significant, multi-stage effort that includes modeling, implementation, and production to deliver business value and to solve real-world problems. Numerous alternatives and customization options are available at each step, and these make it increasingly challenging to prepare an ML model for production within the constraints of your resources and budget. Over the past few years at Amazon Web Services (AWS), our Data Science team has worked with different industry sectors on ML initiatives. We identified pain points shared by many AWS customers, which originate from both organizational problems and technical challenges, and we have developed an optimal approach for delivering production-ready ML solutions.

This guide is for data scientists and ML engineers who are involved in ML pipeline implementations. It describes our approach for delivering production-ready ML pipelines. The guide discusses how you can transition from running ML models interactively (during development) to deploying them as a part of a pipeline (during production) for your ML use case. For this purpose, we have also developed a set of example templates (see the ML Max project), to accelerate the delivery of custom ML solutions to production, so you can get started quickly without having to make too many design choices.

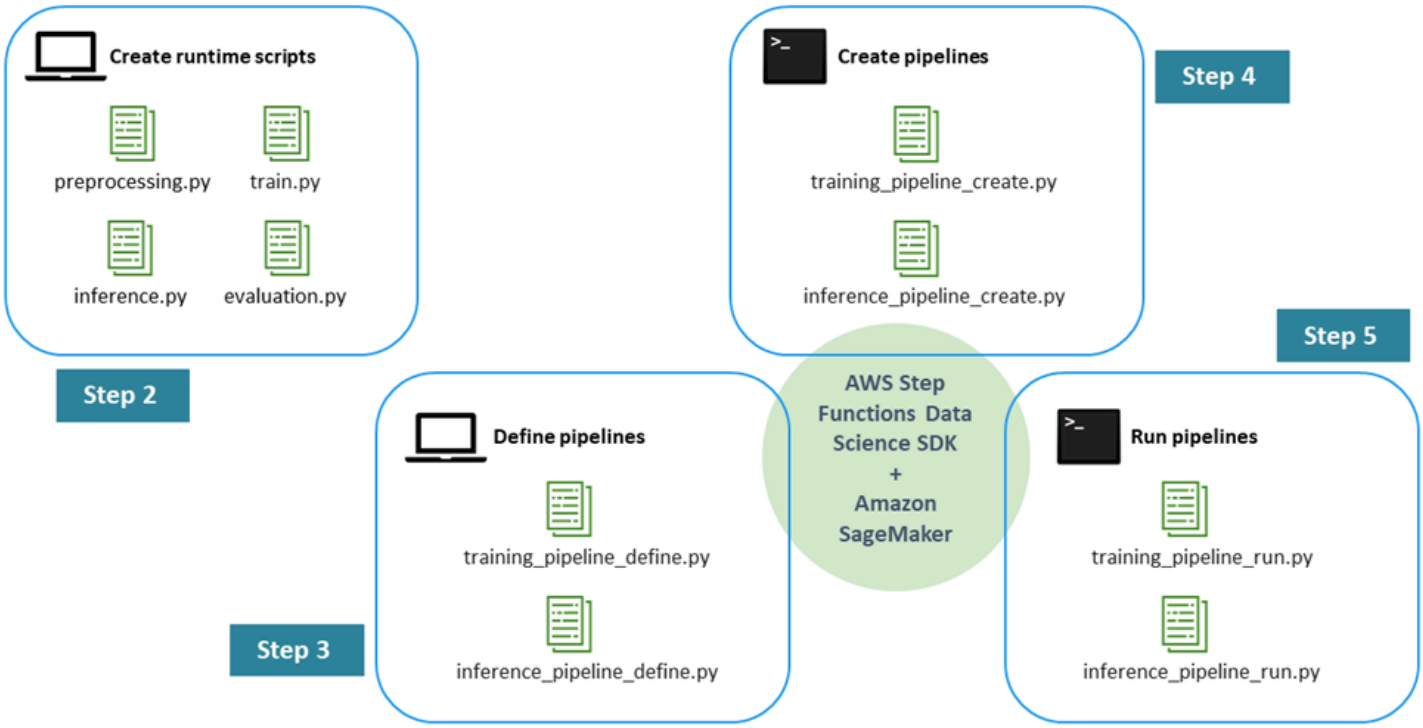
Overview

The process for creating a production-ready ML pipeline consists of the following steps:

- **Step 1. Perform EDA and develop the initial model –** Data scientists make raw data available in Amazon Simple Storage Service (Amazon S3), perform exploratory data analysis (EDA), develop the initial ML model, and evaluate its inference performance. You can conduct these activities interactively through Jupyter notebooks.
- **Step 2. Create the runtime scripts –** You integrate the model with runtime Python scripts so that it can be managed and provisioned by an ML framework (in our case, Amazon SageMaker). This is the first step in moving away from the interactive development of a standalone model toward production. Specifically, you define the logic for preprocessing, evaluation, training, and inference separately.

- **Step 3. Define the pipeline –** You define the input and output placeholders for each step of the pipeline. Concrete values for these will be supplied later, during runtime (step 5). You focus on pipelines for training, inference, cross-validation, and back-testing.
- **Step 4. Create the pipeline –** You create the underlying infrastructure, including the AWS Step Functions state machine instance in an automated (nearly one-click) fashion, by using AWS CloudFormation.
- **Step 5. Run the pipeline –** You run the pipeline defined in step 4. You also prepare metadata and data or data locations to fill in concrete values for the input/output placeholders that you defined in step 3. This includes the runtime scripts defined in step 2 as well as model hyperparameters.
- **Step 6. Expand the pipeline –** You implement continuous integration and continuous deployment (CI/CD) processes, automated retraining, scheduled inference, and similar extensions of the pipeline.

The following diagram illustrates the major steps in this process.



Steps for creating the training and inference pipeline

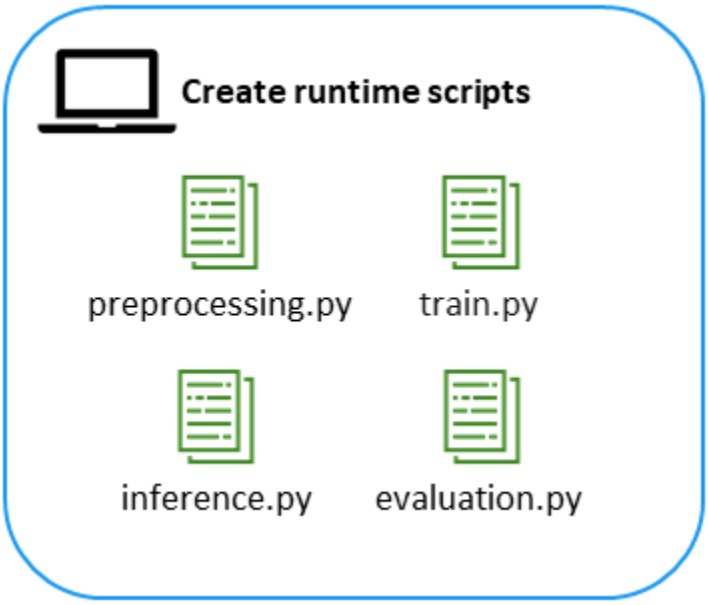
Step 1. Perform EDA and develop the initial model

In this step, data scientists perform exploratory data analysis (EDA) in order to understand the ML use case and data. They then develop the ML models (for example, classification and regression models) to solve the problem in a given use case. During model development, the data scientist often makes assumptions about inputs and outputs, such as data formats, the data lifecycle, and locations of intermediate output. These assumptions should be documented so that they can be used for verification during unit tests in step 2.

Although this step focuses on model development, data scientists often have to write a minimum amount of helper code for preprocessing, training, evaluation, and inference. The data scientist should be able to run this code in the development environment. We also recommend providing optional runtime arguments so that this helper code can be dynamically configured to run in other environments without extensive manual changes. This will accelerate the integration between the model and the pipeline in steps 2 and 3. For example, code for reading the raw data should be encapsulated in functions so that data can be preprocessed in a consistent manner.

We recommend that you start with a framework such as scikit-learn, XGBoost, PyTorch, Keras, or TensorFlow to develop the ML model and its helper code. For example, scikit-learn is a a free ML library that's written in Python. It provides a uniform API convention for objects, and includes four main objects—*estimator*, *predictor*, *transformer,* and *model*—that cover lightweight data transforms, support label and feature engineering, and encapsulate preprocessing and modeling steps. These objects help avoid boilerplate code proliferation and prevent validation and test data from leaking into the training dataset. Similarly, every ML framework has its own implementation of key ML artifacts, and we recommend that you comply with the API conventions of your selected framework when you develop ML models.

Step 2. Create the runtime scripts



In this step, you integrate the model that you developed in step 1 and its associated helper code into an ML platform for production-ready training and inference. Specifically, this involves the development of runtime scripts so that the model can be incorporated into Amazon SageMaker. These standalone Python scripts include predefined SageMaker callback functions and environment variables. They are run inside a SageMaker container that is hosted on an Amazon Elastic Compute Cloud (Amazon EC2) instance. The Amazon SageMaker Python SDK documentation provides detailed information about how these callbacks and auxiliary setup work together for training and inference. (For example, see Prepare a scikit-learn training script and Deploy a scikit-learn model in the SageMaker documentation.) The following sections provide additional recommendations for developing ML runtime scripts, based on our experience working with AWS customers.

Using processing jobs

SageMaker provides two options for performing batch-mode model inference. You can use a SageMaker *processing job* or a *batch transform job*. Each option has advantages and disadvantages.

A processing job consists of a Python file that runs inside a SageMaker container. The processing job consists of whatever logic you put in your Python file. It has these advantages:

- When you understand the basic logic of a training job, processing jobs are straightforward to set up and easy to understand. They share the same abstractions as training jobs (for example, tuning the instance count and data distribution).
- Data scientists and ML engineers have full control over data manipulation options.
- The data scientist doesn't have to manage any I/O component logic except for familiar read/write functionality.
- It's somewhat easier to run the files in non-SageMaker environments, which aids rapid development and local testing.
- If there is an error, a processing job fails as soon as the script fails, and there is no unexpected waiting for retry.

On the other hand, batch transform jobs are an extension of the concept of a SageMaker endpoint. At runtime, these jobs import callback functions, which then handle the I/O for reading the data, loading the model, and making the predictions. Batch transform jobs have these advantages:

- They use a data distribution abstraction that differs from the abstraction used by training jobs.
- They use the same core file and function structure for both batch inference and realtime inference, which is convenient.
- They have a built-in, retry-based fault-tolerance mechanism. For example, if an error occurs on a batch of records, it will retry multiple times before the job is terminated as a failure.

Because of its transparency, its ease of use in multiple environments, and its shared abstraction with training jobs, we decided to use the processing job instead of the batch transform job in the reference architecture presented in this guide.

You should run Python runtime scripts locally before you deploy them in the cloud. Specifically, we recommend that you use the main guard clause when you structure your Python scripts, and perform unit testing.

Using the main guard clause

Use a main guard clause to support module import and to run your Python script. Running Python scripts individually is beneficial for debugging and isolating issues in the ML pipeline. We recommend the following steps:

- Use an argument parser in the Python processing files to specify input/output files and their locations.
- Provide a main guide and test functions for each Python file.
- After you test a Python file, incorporate it into the different stages of the ML pipeline, whether you're using an AWS Step Functions model or a SageMaker processing job.
- Use **Assert** statements in critical sections of the script to facilitate testing and debugging. For example, you can use an **Assert** statement to ensure that the number of dataset features is consistent after loading.

Unit testing

Unit testing of runtime scripts that were written for the pipeline is an important task that's frequently ignored in ML pipeline development. This is because machine learning and data science are relatively new fields and have been slow to adopt well-established software engineering practices such as unit testing. Because the ML pipeline will be used in the production environment, it is essential to test the pipeline code before applying the ML model to real-world applications.

Unit testing the runtime script also provides the following unique benefits for ML models:

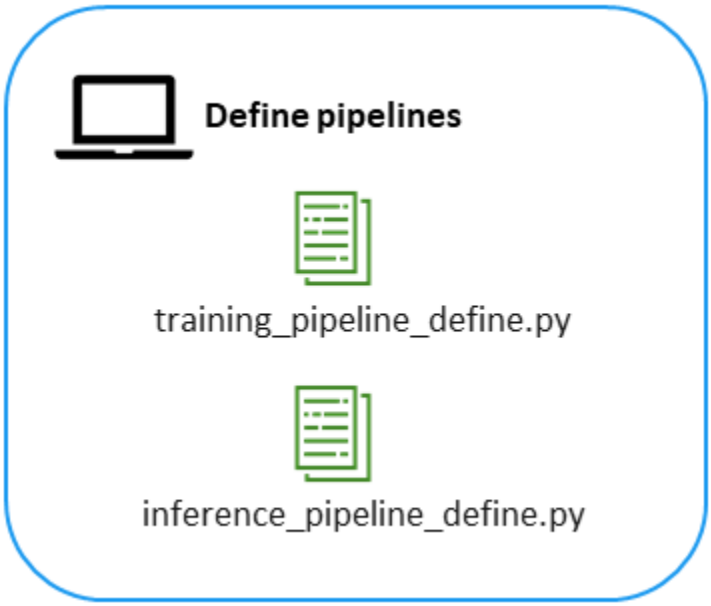
- It prevents unexpected data transformations. Most ML pipelines involve many data transformations, so it's critical for these transformations to perform as expected.
- It validates the reproducibility of the code. Any randomness in the code can be detected by unit testing with different use cases.
- It enforces the modularity of the code. Unit tests are usually associated with the *test coverage* measure, which is the degree to which a particular test suite (a collection of test cases) runs the source code of a program. To achieve high test coverage, developers modularize the code, because it's difficult to write unit tests for a large amount of code without breaking it down into functions or classes.
- It prevents low-quality code or errors from being introduced into production.

We recommend that you use a mature unit testing framework such as pytest to write the unit test cases, because it is easier to manage extensive unit tests within a framework.

Important

Unit testing cannot guarantee that all corner cases are tested, but it can help you proactively avoid mistakes before you deploy the model. We recommend that you also monitor the model after deployment, to ensure operational excellence.

Step 3. Define the pipeline



In this step, the sequence and logic of actions that the pipeline will perform are defined. This includes discrete steps as well as their logical inputs and outputs. For example, what is the state of the data at the beginning of the pipeline? Does it come from multiple files that are at different levels of granularity or from a single flat file? If the data comes from multiple files, do you need a single step for all files or separate steps for each file to define the preprocessing logic? The decision depends on the complexity of the data sources and the extent to which they are preprocessed. In our reference implementation, we use AWS Step Functions, which is a serverless function orchestrator, to define the workflow steps. However, the ML Max framework also supports other pipeline or state machine systems such as Apache AirFlow (see the Different engines for pipeline orchestration section) to drive the development and deployment of ML pipelines.

Using the Step Functions SDK

To define the ML pipeline, we first use the high-level Python API provided by the AWS Step Functions Data Science SDK (the Step Functions SDK) to define two key components of the pipeline: *steps* and *data*. If you think of a pipeline as a directed acyclic graph (DAG), steps represent the nodes on the graph, and data is shown as directed edges that connect one node (step) to the next. Typical examples of ML steps include preprocessing, training, and evaluation. The Step Functions SDK provides a number of built-in steps (such as the TrainingStep) that you can use.

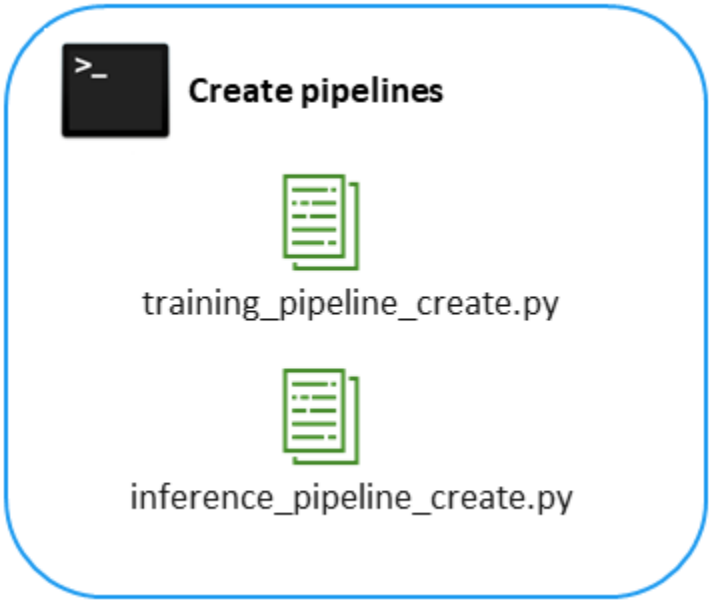
Examples of data include input, output, and many intermediate datasets that are produced by some steps in the pipeline. When you're designing an ML pipeline, you don't know the concrete values of the data items. You can define *data placeholders* that serve as a template (similar to function parameters) and contain only the name of the data item and primitive data types. In this way, you can design a complete pipeline blueprint without knowing the concrete values of data traveling on the graph in advance. For this purpose, you can use the placeholder classes in the Step Functions SDK to explicitly model these data templates. ML pipelines also require configuration parameters to perform fine-grained control over the behavior of each ML step. These special data placeholders are called *parameter placeholders.* Many of their values are unknown when you're defining the pipeline. Examples of parameter placeholders include infrastructure-related parameters that you define during pipeline design (for example, AWS Region or container image URL) and ML modeling-related parameters (such as hyperparameters) that you define when you run the pipeline.

Extending the Step Functions SDK

In our reference implementation, one requirement was to separate ML pipeline definitions from concrete ML pipeline creation and deployment by using specific parameter settings. However, some of the built-in steps in the Step Functions SDK didn't allow us to pass in all these placeholder parameters. Instead, parameter values were expected to be directly obtained during pipeline design time through SageMaker configuration API calls. This works fine if the SageMaker design-time environment is identical to the SageMaker runtime environment, but this is rarely the case in real-world settings. Such a tight coupling between pipeline design-time and runtime, and the assumption that the ML platform infrastructure will remain constant, significantly hinders the applicability of the designed pipeline. In fact, the ML pipeline breaks immediately when the underlying deployment platform undergoes even the slightest change.

To overcome this challenge and produce a robust ML pipeline (which we wanted to design once and run anywhere), we implemented our own custom steps by extending some of the built-in steps, including **TrainingStep**, **ModelStep**, and **TransformerStep**. These extensions are provided in the ML Max project. The interfaces of these custom steps support far more parameter placeholders that can be populated with concrete values during pipeline creation or when the pipeline is running.

Step 4. Create the pipeline



After you define the pipeline logically, it's time to create the infrastructure to support the pipeline. This step requires the following capabilities, at a minimum:

- Storage, to host and manage pipeline inputs and outputs, including code, model artifacts, and data used in training and inference runs.
- Compute (GPU or CPU), for modeling and inference as well as data preprocessing and postprocessing.
- Orchestration, to manage the resources being used and to schedule any regular runs. For example, the model might be retrained on a periodic basis as new data becomes available.
- Logging and alerting, to monitor the pipeline model accuracy, for resource utilization, and for troubleshooting.

Implementation with AWS CloudFormation

To create the pipeline we used AWS CloudFormation, which is an AWS service for deploying and managing infrastructure as code. The AWS CloudFormation templates include the Step Functions definition that was created in the previous step with the Step Functions SDK. This step includes the creation of the AWS-managed Step Functions instance, which is called the *Step Functions state*

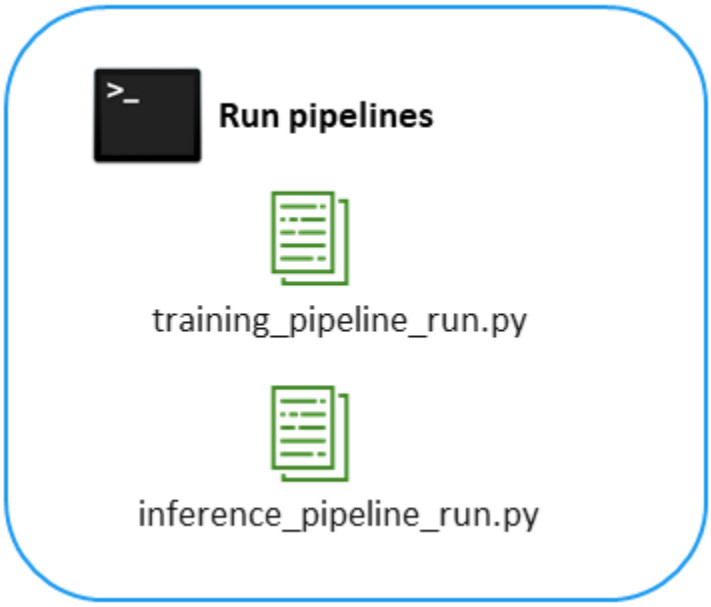
machine. No resources for training and inference are created at this stage, because training and and inference jobs run on demand, only when they're needed, as Amazon SageMaker jobs. This step also includes creating AWS Identity and Access Management (IAM) roles to run the Step Functions, run SageMaker, and read and write from Amazon S3.

Modifying the output from the Step Functions SDK

We had to make some minor modifications to the AWS CloudFormation output from the previous section. We used simple Python string matching to do the following:

- We added logic for creating the Parameters section of the AWS CloudFormation template. This is because we want to create two roles and define the pipeline name as a parameter along with the deployment environment. This step also covers any additional resources and roles that you might want to create, as discussed in step 6.
- We reformatted three fields to have the required `!Sub` prefix and quotation marks so that they can be updated dynamically as a part of the deployment process:
  - The `StateMachineName` property, which names the state machine.
  - The `DefinitionString` property, which defines the state machine.
  - The `RoleArn` property, which is returned by the state machine.

Step 5. Run the pipeline



This step runs the training or inference pipeline that was created in the AWS CloudFormation stacks in step 4. The pipeline can't be run until its internal placeholder parameters have been populated with concrete values. This action of assigning values to placeholder parameters is the primary activity of step 5. Example placeholder parameters include:

- The location of input, output, and intermediate datasets
- The Amazon S3 location of the runtime scripts and other preprocessing or evaluation code that was developed in step 2 (for example, sm_submit_url for the training pipeline)
- The name of the AWS Region

You have to make sure that these path values point to valid data or code before you run the pipeline. For example, if you populate the placeholder parameter that represents the Amazon S3 URL of the Python runtime scripts, you must upload those scripts to that URL. The person who runs the pipeline is responsible for consistency checking and data uploading. People who define or create the pipeline do not have to worry about any of this. Depending on the maturity of the pipeline, this step may be automated to run on a regular (weekly or monthly) basis. Automation also requires robust monitoring, which is an important area but outside the scope of this guide. For the training pipeline run, it would be appropriate to monitor evaluation metrics. For the inference pipeline, it would be appropriate to monitor the input data distribution drift, and, if possible,

collect labels periodically and measure the drift in prediction accuracy. These records from the training and inference runs should be recorded in a database for analysis at a later point in time.

Step 6. Expand the pipeline

This guide explains how you can get started building ML pipelines on AWS quickly, with concrete architecture. There are additional considerations for maturing the pipeline, such as metadata management, experiment tracking, and monitoring. These are important topics that are outside the scope of this guide. The following sections discuss another aspect of pipeline management, which is pipeline automation.

Different levels of automation

Although you can set up a training pipeline manually in the AWS SageMaker console, in practice, we recommend minimizing manual touchpoints in the deployment of ML training pipelines to ensure that ML models are deployed consistently and repeatedly. Depending on your requirements and the business problems you're addressing, you can determine and implement a deployment strategy on three levels: semi-automated, fully automated, and fully managed.

- Semi-automated — By default, the steps discussed in the previous section follow a semi-automated approach, because they deploy the training and inference pipeline by using AWS Cloudformation templates. This helps ensure the reproducibility of the pipeline and helps you change and update it easily.
- Fully automated — A more advanced option is to use continuous integration and continuous deployment (CI/CD) to the development, staging, and production environments. Incorporating CI/CD practices to the deployment of the training pipeline can ensure that automation includes traceability as well as quality gates.
- Fully managed — Ultimately, you can develop a fully managed system so that you can deploy a ML training pipeline with a set of simple manifests, and the system can self-configure and coordinate required AWS services.

In this guide, we chose to present a concrete architecture. However, there are alternative technologies you can consider. The next two sections discuss some alternative choices for the platform and the orchestration engine.

Different platforms for ML workloads

AWS SageMaker is the AWS managed service for training and serving ML models. Many users appreciate its wide array of built-in features and the many options it offers for running ML

workloads. SageMaker is particularly useful if you are just getting started with implementing ML in the cloud. The key features of SageMaker include:

- Built-in traceability (including labeling, training, model tracking, optimization, and inference).
- Built-in one-click options for training and inference with minimal Python and ML experience.
- Advanced hyperparameter tuning.
- Support for all major artificial intelligence and machine learning (ML/AI) frameworks and custom Docker containers.
- Built-in monitoring capabilities.
- Built-in tracking of histories, including training jobs, processing jobs, batch transform jobs, models, endpoints, and searchability. Some histories, such as training, processing, and batch transform, are immutable and append-only.

One of the alternatives to using Amazon SageMaker is AWS Batch. AWS Batch provides a lower level of control over the compute and orchestration for your environment, but it isn't custom-built for machine learning. Some of its key features include:

- Out-of-the-box automatic scaling of compute resources based on workload.
- Out-of-the-box support for job priority, retries, and job dependencies.
- Queue-based approach that supports building recurrent and on-demand jobs.
- Support for CPU and GPU workloads. The ability to use GPU for building ML models is critical, because GPU can speed up the training process significantly, especially for deep learning models.
- Ability to define a custom Amazon Machine Image (AMI) for the compute environment.

Different engines for pipeline orchestration

The second main component is the pipeline orchestration layer. AWS provides Step Functions for a fully managed orchestration experience. A popular alternative to Step Functions is Apache Airflow. When making a decision between the two, consider the following:

- Required infrastructure – AWS Step Functions is a fully managed service and is serverless, whereas Airflow requires managing your own infrastructure and is based on open-source software. As a result, Step Functions provides high availability out of the box, whereas administering Apache Airflow requires additional steps.
- Scheduling capabilities – Both Step Functions and Airflow provide comparable functionalities.

- Visualization capabilities and UI – Both Step Functions and Airflow provide comparable functionalities.
- Passing variables within the computational graph – Step Functions provides limited functionality for using AWS Lambda functions, whereas Airflow provides XCom interfaces.
- Usage – Step Functions is very popular among AWS customers, and Airflow has been widely adopted by the data engineering community.

Conclusion

As machine learning transitions from a research discipline to an applied field, we've seen a yearly growth of 25 percent in ML pipeline development, deployment, and operation in various industries. The business value of ML is realized through day-to-day ML operations and pipelines, which, in turn, drive the research and development of ML models and algorithms. Nonetheless, deploying ML in production presents numerous challenges, because it interweaves significantly different activities and artifacts, such as data management, processing, analysis, modeling, verification, and security. Through numerous AI/ML engagements with AWS customers, our Data Science team has observed that a key challenge is the lack of an end-to-end workflow that would provide a set of templates for optimally fusing or separating different ML DevOps activities and artifacts. In this guide, we presented the ML Max workflow to address this pressing issue. ML Max provides step-by-step guidelines and a set of programming templates. The goal is to enable a fast and cost-effective transition from an interactive model development phase to a complete, scalable ML pipeline configuration that is ready for production.

Resources

Related guides and patterns

- Migrate ML Build, Train, and Deploy workloads to Amazon SageMaker AI using AWS Developer Tools
- AWS Prescriptive Guidance website

AWS services

- AWS Batch
- AWS CloudFormation
- AWS Identity and Access Management (IAM)
- Amazon SageMaker AI
- AWS Step Functions

Document history

The following table describes significant changes to this guide.

Change	Description	Date
Initial publication	—	January 29, 2021